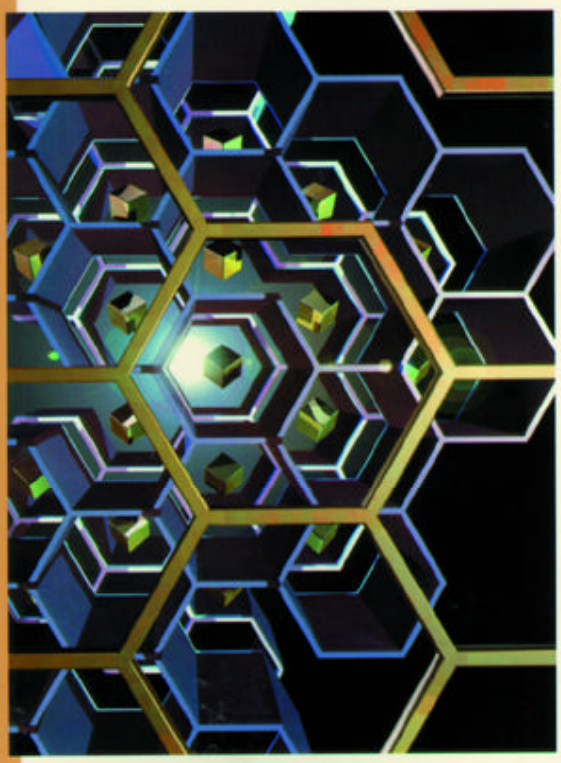
WEBMASTER SERIES

JAVA™ DATABASE PROGRAMMING WITH JDBC™



- Master the creation of Java™ databases with JavaSoft's JDBC™ API
- Features in-depth coverage of ODBC, SQL, database security, and more

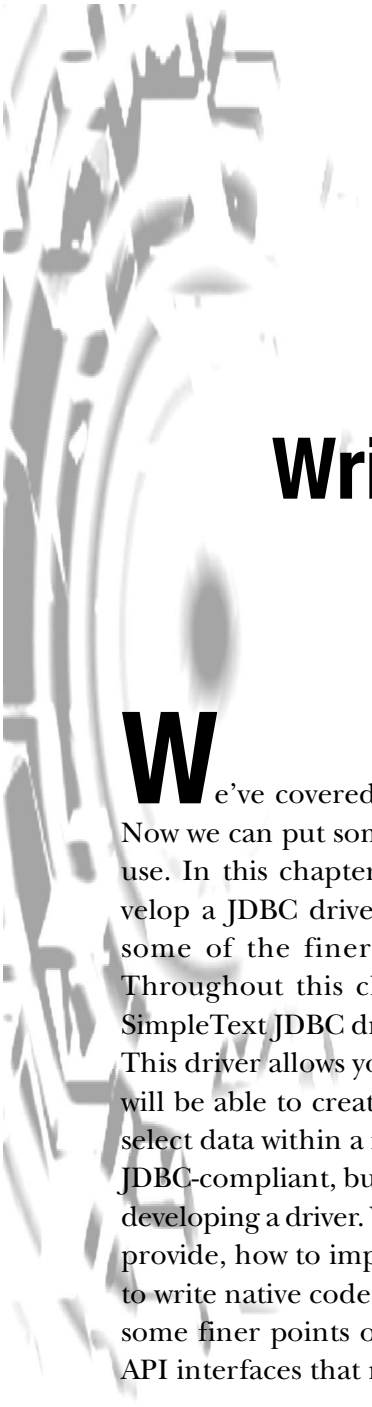
CD-ROM INCLUDED

Pratik Patel & Karl Moss

CORIOLIS GROUP BOOKS

Chapter 10

Writing Database Drivers



CHAPTER

10

Writing Database Drivers

We've covered a lot of territory so far in this book. Now we can put some of your newly gained knowledge to use. In this chapter, we will explore what it takes to develop a JDBC driver. In doing so, we will also touch on some of the finer points of the JDBC specification. Throughout this chapter, I will use excerpts from the SimpleText JDBC driver that is included on the CD-ROM. This driver allows you to manipulate simple text files; you will be able to create and drop files, as well as insert and select data within a file. The SimpleText driver is not fully JDBC-compliant, but it provides a strong starting point for developing a driver. We'll cover what the JDBC components provide, how to implement the JDBC API interfaces, how to write native code to bridge to an existing non-Java API, some finer points of driver writing, and the major JDBC API interfaces that must be implemented.

The JDBC Driver Project: SimpleText

The SimpleText JDBC driver is just that—a JDBC driver that manipulates simple text files, with a few added twists.

It is not a full-blown relational database system, so I would not recommend attempting to use it as one. If you are looking for a good way to prototype a system, or need a very lightweight database system to drive a simplistic application or applet, then SimpleText is for you. More importantly, though, the SimpleText driver can serve as a starting point for your own JDBC driver. Before continuing, let's take a look at the SimpleText driver specifications.

SimpleText SQL Grammar

The SimpleText JDBC driver supports a very limited SQL grammar. This is one reason that the driver is not JDBC compliant; a JDBC-compliant driver must support ANSI92 entry level SQL grammar. The following SQL statements define the base SimpleText grammar:

create-table-statement ::= CREATE TABLE *table-name*

(*column-element* [, *column-element*]...)

drop-table-statement ::= DROP TABLE *table-name*

insert-statement ::= INSERT INTO *table-name*

[(*column-identifier* [, *column-identifier*]...)] VALUES

(*insert-value* [, *insert-value*]...)

select-statement ::= SELECT *select-list* FROM *table-name* [WHERE *search-condition*]

The following elements are used in these SQL statements:

column-element ::= *column-identifier data-type*

column-identifier ::= *user-defined-name*

comparison-operator ::= < | > | = | <>

data-type ::= VARCHAR | NUMBER | BINARY

dynamic-parameter ::= ?

insert-value ::= *dynamic-parameter* | *literal*

search-condition ::= *column-identifier comparison-operator literal*

select-list := * | *column-identifier* [, *column-identifier*]...

table-name := *user-defined-name*

user-defined-name := *letter* [*digit* | *letter*]...

What all this grammar means is that the SimpleText driver supports a **CREATE TABLE** statement, a **DROP TABLE** statement, an **INSERT** statement (with parameters), and a very simple **SELECT** statement (with a **WHERE** clause). It may not seem like much, but this grammar is the foundation that will allow us to create a table, insert some data, and select it back.

SimpleText File Format

The format of the files used by the SimpleText driver is, of course, very simple. The first line contains a signature, followed by each one of the column names (and optional data types). Any subsequent lines in the text file are assumed to be comma-separated data. There is no size limit to the text file, but the larger the file, the longer it takes to retrieve data (the entire file is read when selecting data; there is no index support). The data file extension is hard coded to be .SDF (Simple Data File). For example, the statement

```
CREATE TABLE TEST (COL1 VARCHAR, COL2 NUMBER, COL3 BINARY)
```

creates a file named TEST.SDF, with the following initial data:

```
.SDFCOL1,#COL2,@COL3
```

Note that none of the SQL grammar is case-sensitive. The .SDF is the file signature (this is how the SimpleText driver validates whether the text file can be used), followed by a comma-separated list of column names. The first character of the column name can specify the data type of the column. A column name starting with a # indicates a numeric column, while a column name starting with an @ indicates a binary column. What's that? Binary data in a text file? Well, not quite. A binary column actually contains an offset pointer into a sister file. This file, with an extension of .SBF (Simple Binary File), contains any binary data for columns in the text file, as well as the length of the data (maximum length of 1048576 bytes). Any

other column name is considered to be character data (with a maximum length of 5120 bytes). The following statement shows how data is inserted into the TEST table:

```
INSERT INTO TEST VALUES ('FOO', 123, '0123456789ABCDEF')
```

After the INSERT, TEST.SDF will contain the following data:

```
.SDFCOL1,#COL2,@COL3
FOO,123,0
```

COL3 contains an offset of zero since this is the first row in the file. This is the offset from within the TEST.SBF table in which the binary data resides. Starting at the given offset, the first four bytes will be the length indicator, followed by the actual binary data that was inserted. Note that any character or binary data must be enclosed in single quotation marks.

We'll be looking at plenty of code from the SimpleText driver throughout this chapter. But first, let's start by exploring what is provided by the JDBC developer's kit.

The DriverManager

The JDBC **DriverManager** is a static class that provides services to connect to JDBC drivers. The **DriverManager** is provided by JavaSoft and does not require the driver developer to perform any implementation. Its main purpose is to assist in loading and initializing a requested JDBC driver. Other than using the **DriverManager** to register a JDBC driver (**registerDriver**) to make itself known and to provide the logging facility (which is covered in detail later), a driver does not interface with the **DriverManager**. In fact, once a JDBC driver is loaded, the **DriverManager** drops out of the picture all together, and the application or applet interfaces with the driver directly.

JDBC Exception Types

JDBC provides special types of exceptions to be used by a driver: **SQLException**, **SQLWarning**, and **DataTruncation**. The **SQLException** class is the foundation for the other types of JDBC exceptions, and extends

java.lang.Exceptn. When created, an **SQLException** can have three pieces of information: a **String** describing the error, a **String** containing the XOPEN SQLstate (as described in the XOPEN SQL specification), and an **int** containing an additional vendor or database-specific error code. Also note that **SQLExceptions** can be chained together; that is, multiple **SQLExceptions** can be thrown for a single operation. The following code shows how an **SQLException** is thrown:

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// fooBar
// Demonstrates how to throw an SQLException
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
public void fooBar()
    throws SQLException
{
    throw new SQLException("I just threw a SQLException");
}
```

Here's how you call fooBar and catch the **SQLException**:

```
try {
    fooBar();
}
catch (SQLException ex) {

    // If an SQLException is thrown, we'll end up here. Output the error
    // message, SQLstate, and vendor code.
    System.out.println("A SQLException was caught!");
    System.out.println("Message: " + ex.getMessage());
    System.out.println("SQLState: " + ex.getSQLState());
    System.out.println("Vendor Code: " + ex.getErrorCode());
}
```

An **SQLWarning** is similar to an **SQLException** (it extends **SQLException**). The main difference is in semantics. If an **SQLException** is thrown, it is considered to be a critical error (one that needs attention). If an **SQLWarning** is thrown, it is considered to be a non-critical error (a warning or informational message). For this reason, JDBC treats **SQLWarnings** much differently than **SQLExceptions**. **SQLExceptions** are thrown just like any other type of exception; **SQLWarnings** are not thrown, but put on a list of warnings on an owning object type (for instance, **Connection, State-**

ment, or **ResultSet**, which we'll cover later). Because they are put on a list, it is up to the application to poll for warnings after the completion of an operation. Listing 10.1 shows a method that accepts an **SQLWarning** and places it on a list.

Listing 10.1 Placing an SQLWarning on a list.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// setWarning
// Sets the given SQLWarning in the warning chain. If null, the
// chain is reset. The local attribute lastWarning is used
// as the head of the chain.
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
protected void setWarning(
    SQLWarning warning)
{
    // A null warning can be used to clear the warning stack
    if (warning == null) {
        lastWarning = null;
    }
    else {
        // Set the head of the chain. We'll use this to walk through the
        // chain to find the end.
        SQLWarning chain = lastWarning;

        // Find the end of the chain. When the current warning does
        // not have a next pointer, it must be the end of the chain.
        while (chain.getNextWarning() != null) {
            chain = chain.getNextWarning();
        }

        // We're at the end of the chain. Add the new warning
        chain.setNextWarning(warning);
        }
}
```

Listing 10.2 uses this method to create two **SQLWarnings** and chain them together.

Listing 10.2 Chaining SQLWarnings together.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// fooBar
// Do nothing but put two SQLWarnings on our local
```

```
// warning stack (lastWarning).
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
protected void fooBar()
{
    // First step should always be to clear the stack. If a warning
    // is lingering, it will be discarded. It is up to the application to
    // check and clear the stack.
    setWarning(null);

    // Now create our warnings
    setWarning(new SQLWarning("Warning 1"));
    setWarning(new SQLWarning("Warning 2"));
}
```

Now we'll call the method that puts two **SQLWarnings** on our warning stack, then poll for the warning using the JDBC method **getWarnings**, as shown in Listing 10.3.

Listing 10.3 Polling for warnings.

```
// Call fooBar to create a warning chain
fooBar();

// Now, poll for the warning chain. We'll simply dump any warning
// messages to standard output.
SQLWarning chain = getWarnings();

if (chain != null) {
    System.out.println("Warning(s):");

    // Display the chain until no more entries exist
    while (chain != null) {
        System.out.println("Message: " + chain.getMessage());

        // Advance to the next warning in the chain. null will be
        // returned if no more entries exist.
        chain = chain.getNextWarning();
    }
}
```

DataTruncation objects work in the same manner as **SQLWarnings**. A **DataTruncation** object indicates that a data value that was being read or written was truncated, resulting in a loss of data. The **DataTruncation** class has attributes that can be set to specify the column or parameter number,

whether a truncation occurred on a read or a write, the size of the data that should have been transferred, and the number of bytes that were actually transferred. We can modify our code from Listing 10.2 to include the handling of **DataTruncation** objects, as shown in Listing 10.4.

Listing 10.4 Creating dDataTruncation warnings.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// fooBar
// Do nothing but put two SQLWarnings on our local
// warning stack (lastWarning) and a DataTruncation
// warning.
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
protected void fooBar()
{
    // First step should always be to clear the stack. If a warning
    // is lingering, it will be discarded. It is up to the application to
    // check and clear the stack.
    setWarning(null);

    // Now create our warnings
    setWarning(new SQLWarning("Warning 1"));
    setWarning(new SQLWarning("Warning 2"));

    // And create a DataTruncation indicating that a truncation
    // occurred on column 1, 1000 bytes were requested to
    // read, and only 999 bytes were read.
    setWarning(new DataTruncation(1, false, true, 1000, 999);
}
```

Listing 10.5 shows the modified code to handle the **DataTruncation**.

Listing 10.5 Processing DataTruncation warnings.

```
// Call fooBar to create a warning chain
fooBar();

// Now, poll for the warning chain. We'll simply dump any warning
// messages to standard output.
SQLWarning chain = getWarnings();

if (chain != null) {
    System.out.println("Warning(s):");
```

```
    // Display the chain until no more entries exist
    while (chain != null) {
        // The only way we can tell if this warning is a DataTruncation
        // is to attempt to cast it. This may fail, indicating that
        // it is just an SQLWarning.
        try {
            DataTruncation trunc = (DataTruncation) chain;
            System.out.println("Data Truncation on column: " +
                trunc.getIndex());
        }
        catch (Exception ex) {
            System.out.println("Message: " + chain.getMessage());
        }

        // Advance to the next warning in the chain. null will be
        // returned if no more entries exist.
        chain = chain.getNextWarning();
    }
}
```

JDBC Data Types

The JDBC specification provides definitions for all of the SQL data types that can be supported by a JDBC driver. Only a few of these data types may be natively supported by a given database system, which is why data coercion becomes such a vital service (we'll discuss data coercion a little later in this chapter). The data types are defined in **Types.class**:

```
public class Types
{
    public final static int BIT = -7;
    public final static int TINYINT = -6;
    public final static int SMALLINT = 5;
    public final static int INTEGER = 4;
    public final static int BIGINT = -5;
    public final static int FLOAT = 6;
    public final static int REAL = 7;
    public final static int DOUBLE = 8;
    public final static int NUMERIC = 2;
    public final static int DECIMAL = 3;
    public final static int CHAR = 1;
    public final static int VARCHAR = 12;
    public final static int LONGVARCHAR = -1;
    public final static int DATE = 91;
```

```
    public final static int TIME = 92;
    public final static int TIMESTAMP = 93;
    public final static int BINARY = -2;
    public final static int VARBINARY = -3;
    public final static int LONGVARBINARY = -4;
    public final static int OTHER = 1111;
}
```

At a minimum, a JDBC driver must support one (if not all) of the character data types (**CHAR**, **VARCHAR**, and **LONGVARCHAR**). A driver may also support driver-specific data types (**OTHER**) which can only be accessed in a JDBC application as an **Object**. In other words, you can get data as some type of object and put it back into a database as that same type of object, but the application has no idea what type of data is actually contained within. Let's take a look at each of the data types more closely.

Character Data: CHAR, VARCHAR, And LONGVARCHAR

CHAR, **VARCHAR**, and **LONGVARCHAR** data types are used to express character data. These data types are represented in JDBC as Java **String** objects. Data of type **CHAR** is represented as a fixed-length **String**, and may include some padding spaces to ensure that it is the proper length. If data is being written to a database, the driver must ensure that the data is properly padded. Data of type **VARCHAR** is represented as a variable-length **String**, and is trimmed to the actual length of the data. **LONGVARCHAR** data can be either a variable-length **String** or returned by the driver as a Java **InputStream**, allowing the data to be read in chunks of whatever size the application desires.

Exact Numeric Data: NUMERIC And DECIMAL

The **NUMERIC** and **DECIMAL** data types are used to express signed, exact numeric values with a fixed number of decimal places. These data types are often used to represent currency values. **NUMERIC** and **DECIMAL** data are both represented in JDBC as **Numeric** objects. The **Numeric** class is new with JDBC, and we'll be discussing it shortly.

Binary Data: BINARY, VARBINARY, And LONGVARBINARY

The **BINARY**, **VARBINARY**, and **LONGVARBINARY** data types are used to express binary (non-character) data. These data types are represented

in JDBC as Java byte arrays. Data of type **BINARY** is represented as a fixed-length byte array, and may include some padding zeros to ensure that it is the proper length. If data is being written to a database, the driver must ensure that the data is properly padded. Data of type **VARBINARY** is represented as a variable-length byte array, and is trimmed to the actual length of the data. **LONGVARBINARY** data can either be a variable-length byte array or returned by the driver as a Java **InputStream**, allowing the data to be read in chunks of whatever size the application desires.

Boolean Data: BIT

The **BIT** data type is used to represent a boolean value—either true or false—and is represented in JDBC as a **Boolean** object or **boolean** data type.

Integer Data: TINYINT, SMALLINT, INTEGER, And BIGINT

The **TINYINT**, **SMALLINT**, **INTEGER**, and **BIGINT** data types are used to represent signed integer data. Data of type **TINYINT** is represented in JDBC as a Java *byte* data type (1 byte), with a minimum value of -128 and a maximum value of 127. Data of type **SMALLINT** is represented in JDBC as a Java *short* data type (2 bytes), with a minimum value of -32,768 and a maximum value of 32,767. Data of type **INTEGER** is represented as a Java *int* data type (4 bytes), with a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647. Data of type **BIGINT** is represented as a Java *long* data type (8 bytes), with a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807.

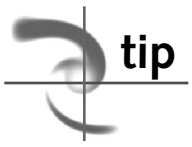
Floating-Point Data: REAL, FLOAT, And DOUBLE

The **REAL**, **FLOAT**, and **DOUBLE** data types are used to represent signed, approximate values. Data of type **REAL** supports seven digits of mantissa precision, and is represented as a Java *float* data type. Data of types **FLOAT** and **DOUBLE** support 15 digits of mantissa precision, and are represented as Java *double* data types.

Time Data: DATE, TIME, And TIMESTAMP

The **DATE**, **TIME**, and **TIMESTAMP** data types are used to represent dates and times. Data of type **DATE** supports specification of the month, day, and year, and is represented as a JDBC **Date** object. Data of type **TIME**

supports specification of the hour, minutes, seconds, and milliseconds, and is represented as a JDBC **Time** object. Data of type **TIMESTAMP** supports specification of the month, day, year, hour, minutes, seconds, and milliseconds, and is represented as a JDBC **Timestamp** object. The **Date**, **Time,** and **Timestamp** objects, which we'll get into a bit later, are new with JDBC.



Be aware of date limitations.
One important note about **Date** and **Timestamp** objects: The Java calendar starts at January 1, 1970, which means that you cannot represent dates prior to 1970.

New Data Classes

The JDBC API introduced several new data classes. These classes were developed to solve specific data-representation problems like how to accurately represent fixed-precision numeric values (such as currency values) for **NUMERIC** and **DECIMAL** data types, and how to represent time data for **DATE**, **TIME**, and **TIMESTAMP** data types.

Numeric

As mentioned before, the **Numeric** class was introduced with the JDBC API to represent signed, exact numeric values with a fixed number of decimal places. This class is ideal for representing monetary values, allowing accurate arithmetic operations and comparisons. Another aspect is the ability to change the rounding value. Rounding is performed if the value of the scale (the number of fixed decimal places) plus one digit to the right of the decimal point is greater than the rounding value. By default, the rounding value is 4. For example, if the result of an arithmetic operation is 2.495, and the scale is 2, the number is rounded to 2.50. Listing 10.6 provides an example of changing the rounding value. Imagine that you are a devious retailer investigating ways to maximize your profit by adjusting the rounding value.

Listing 10.6 Changing the rounding value.

```
import java.sql.*;

class NumericRoundingValueTest {

    public static void main(String args[]) {
```

```
        // Set our price and discount amounts
        Numeric price = new Numeric(4.91, 2);
        Numeric discount = new Numeric(0.15, 2);
        Numeric newPrice;

        // Give the item a discount
        newPrice = discountItem(price, discount);

        System.out.println("discounted price="+newPrice.toString());

        // Now, give the item a discount with a higher rounding value.
        // This will lessen the discount amount in many cases.
        discount.setRoundingValue(9);

        newPrice = discountItem(price, discount);

        System.out.println("discounted price with high rounding="+
                newPrice.toString());
    }

    // Perform the calculation to discount a price
    public static Numeric discountItem(
        Numeric price,
        Numeric discount)
    {
        return price.subtract(price.multiply(discount));
    }
}
```

Listing 10.6 produces the following output:

```
discounted price=004.17
discounted price with high rounding=004.18
```

Date

The **Date** class is used to represent dates in the ANSI SQL format YYYY-MM-DD, where YYYY is a four-digit year, MM is a two-digit month, and DD is a two-digit day. The JDBC **Date** class extends the existing **java.util.Date** class (setting the hour, minutes, and seconds to zero) and, most importantly, adds two methods to convert Strings into dates, and vice-versa:

```
// Create a Date object with a date of June 30th, 1996
Date d = Date.valueOf("1996-06-30");
```

```
// Print the date
System.out.println("Date=" + d.toString());

// Same thing, without leading zeros
Date d2 = Date.valueOf("1996-6-30");
System.out.println("Date=" + d2.toString());
```

The **Date** class also serves very well in validating date values. If an invalid date string is passed to the **valueOf** method, a **java.lang.IllegalArgument-Exception** is thrown:

```
String s;

// Get the date from the user
.
.
.
// Validate the date
try {
    Date d = Date.valueOf(s);
}
catch (java.lang.IllegalArgumentException ex) {
    // Invalid date, notify the application
    .
    .
    .
}
```

It is worth mentioning again that the Java date epoch is January 1, 1970; therefore, you cannot represent any date values prior to January 1, 1970, with a **Date** object.

Time

The **Time** class is used to represent times in the ANSI SQL format HH:MM:SS, where HH is a two-digit hour, MM is a two-digit minute, and SS is a two-digit second. The JDBC **Time** class extends the existing **java.util.Date** class (setting the year, month, and day to zero) and, most importantly, adds two methods to convert Strings into times, and vice-versa:

```
// Create a Time object with a time of 2:30:08 pm
Time t = Time.valueOf("14:30:08");
```

```
// Print the time
System.out.println("Time=" + t.toString());

// Same thing, without leading zeros
Time t2 = Time.valueOf("14:30:8");
System.out.println("Time=" + t2.toString());
```

The **Time** class also serves very well in validating time values. If an invalid time string is passed to the **valueOf** method, a **java.lang.IllegalArgumentException** is thrown:

```
String s;

// Get the time from the user
.
.
.
// Validate the time
try {
    Time t = Time.valueOf(s);
}
catch (java.lang.IllegalArgumentException ex) {
    // Invalid time, notify the application
    .
    .
    .
}
```

Timestamp

The **Timestamp** class is used to represent a combination of date and time values in the ANSI SQL format YYYY-MM-DD HH:MM:SS.F..., where YYYY is a four-digit year, MM is a two-digit month, DD is a two-digit day, HH is a two-digit hour, MM is a two-digit minute, SS is a two-digit second, and F is an optional fractional second up to nine digits in length. The JDBC **Timestamp** class extends the existing **java.util.Date** class (adding the fraction seconds) and, most importantly, adds two methods to convert Strings into timestamps, and vice-versa:

```
// Create a Timestamp object with a date of 1996-06-30 and a time of
// 2:30:08 pm.
Timestamp t = Timestamp.valueOf("1996-06-30 14:30:08");

// Print the timestamp
```

```
System.out.println("Timestamp=" + t.toString());

// Same thing, without leading zeros
Timestamp t2 = Timestamp.valueOf("1996-6-30 14:30:8");
System.out.println("Timestamp=" + t2.toString());
```

The **Timestamp** class also serves very well in validating timestamp values. If an invalid time string is passed to the **valueOf** method, a **java.lang.IllegalArgumentException** is thrown:

```
String s;

// Get the timestamp from the user
.
.
.
// Validate the timestamp
try {
    Timestamp t = Timestamp.valueOf(s);
}
catch (java.lang.IllegalArgumentException ex) {
    // Invalid timestamp, notify the application
    .
    .
    .
}
```

As is the case with the **Date** class, the Java date epoch is January 1, 1970; therefore, you cannot represent any date values prior to January 1, 1970, with a **Timestamp** object.

Native Drivers: You're Not From Around Here, Are Ya?

Before beginning to implement a JDBC driver, the first question that must be answered is: Will this driver be written completely in Java, or will it contain native (machine dependent) code? You may be forced to use native code because many major database systems—such as Oracle, Sybase, and SQLServer—do not provide Java client software. In this case, you will need to write a small library containing C code to bridge from Java to the database client API (the JDBC to ODBC Bridge is a perfect example). The

obvious drawback is that the JDBC driver is not portable and cannot be automatically downloaded by today's browsers.

If a native bridge is required for your JDBC driver, you should keep a few things in mind. First, do as little as possible in the C bridge code; you will want to keep the bridge as small as possible, ideally creating just a Java wrapper around the C API. Most importantly, avoid the temptation of performing memory management in C (i.e. malloc). This is best left in Java code, since the Java Virtual Machine so nicely takes care of garbage collection. Secondly, keep all of the native method declarations in one Java class. By doing so, all of the bridge routines will be localized and much easier to maintain. Finally, don't make any assumptions about data representation. An integer value may be 2 bytes on one system, and 4 bytes on another. If you are planning to port the native bridge code to a different system (which is highly likely), you should provide native methods that provide the size and interpretation of data.

Listing 10.7 illustrates these suggestions. This module contains all of the native method declarations, as well as the code to load our library. The library will be loaded when the class is instantiated.

Listing 10.7 Java native methods.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// MyBridge.java
//
// Sample code to demonstrate the use of native methods
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
package jdbc.test;

import java.sql.*;

public class MyBridge
    extends Object
{
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Constructor
    // Attempt to load our library. If it can't be loaded, an
    // SQLException will be thrown.
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    public MyBridge()
        throws SQLException
    {
        try {
```

```
            // Attempt to load our library. For Win95/NT, this will
            // be myBridge.dll. For Unix systems, this will be
            // libmyBridge.so.
            System.loadLibrary("myBridge");
        }
        catch (UnsatisfiedLinkError e) {
            throw new SQLException("Unable to load myBridge library");
        }
    }

    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Native method declarations
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    // Get the size of an int
    public native int getINTSize();

    // Given a byte array, convert it to an integer value
    public native int getINTValue(byte intValue[]);

    // Call some C function that does something with a String, and
    // returns an integer value.
    public native void callSomeFunction(String stringValue, byte
    intValue[]);
}
```

Once this module has been compiled (javac), a Java generated header file and C file must be created:

```
javah jdbc.test.MyBridge
javah -stubs jdbc.test.MyBridge
```

These files provide the mechanism for the Java and C worlds to communicate with each other. Listing 10.8 shows the generated header file (jdbc_test_MyBridge.h, in this case), which will be included in our C bridge code.

Listing 10.8 Machine-generated header file for native methods.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <native.h>
/* Header for class jdbc_test_MyBridge */

#ifndef _Included_jdbc_test_MyBridge
```

```
#define _Included_jdbc_test_MyBridge

typedef struct Classjdbc_test_MyBridge {
    char PAD; /* ANSI C requires structures to have at least one member */
} Classjdbc_test_MyBridge;
HandleTo(jdbc_test_MyBridge);

#ifdef __cplusplus
extern "C" {
#endif
__declspec(dllexport) long jdbc_test_MyBridge_getINTSize(struct
Hjdbc_test_MyBridge *);
__declspec(dllexport) long jdbc_test_MyBridge_getINTValue(struct
Hjdbc_test_MyBridge *,HArrayOfByte *);
struct Hjava_lang_String;
__declspec(dllexport) void jdbc_test_MyBridge_callSomeFunction(struct
Hjdbc_test_MyBridge *,struct Hjava_lang_String *,HArrayOfByte *);
#ifdef __cplusplus
}
#endif
#endif
```

The generated C file (shown in Listing 10.9) must be compiled and linked with the bridge.

Listing 10.9 Machine-generated C file for native methods.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <StubPreamble.h>

/* Stubs for class jdbc/test/MyBridge */
/* SYMBOL: "jdbc/test/MyBridge/getINTSize()I",
Java_jdbc_test_MyBridge_getINTSize_stub */
__declspec(dllexport) stack_item
*Java_jdbc_test_MyBridge_getINTSize_stub(stack_item *_P_,struct execenv
*_EE_) {
    extern long jdbc_test_MyBridge_getINTSize(void *);
    _P_[0].i = jdbc_test_MyBridge_getINTSize(_P_[0].p);
    return _P_ + 1;
    }
    /* SYMBOL: "jdbc/test/MyBridge/getINTValue([B)I",
    Java_jdbc_test_MyBridge_getINTValue_stub */
    __declspec(dllexport) stack_item
    *Java_jdbc_test_MyBridge_getINTValue_stub(stack_item *_P_,struct
    execenv *_EE_) {
    extern long jdbc_test_MyBridge_getINTValue(void *,void *);
```

```
       _P_[0].i = jdbc_test_MyBridge_getINTValue(_P_[0].p,((_P_[1].p)));
       return _P_ + 1;
}
/* SYMBOL: "jdbc/test/MyBridge/callSomeFunction(Ljava/lang/String;[B)V",
Java_jdbc_test_MyBridge_callSomeFunction_stub */
__declspec(dllexport) stack_item
*Java_jdbc_test_MyBridge_callSomeFunction_stub(stack_item *_P_,struct
execenv *_EE_) {
       extern void jdbc_test_MyBridge_callSomeFunction(void *,void *,void
       *);
       (void) jdbc_test_MyBridge_callSomeFunction(_P_[0].p,((_P_[1].p)),
       ((_P_[2].p)));return _P_;
}
```

The bridge code is shown in Listing 10.10. The function prototypes were taken from the generated header file.

Listing 10.10 Bridge code.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// MyBridge.c
//
// Sample code to demonstrate the use of native methods
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#include <stdio.h>
#include <ctype.h>
#include <string.h>

// Java internal header files
#include "StubPreamble.h"
#include "javaString.h"

// Our header file generated by JAVAH
#include "jdbc_test_MyBridge.h"

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// getINTSize
// Return the size of an int
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
long jdbc_test_MyBridge_getINTSize(
    struct Hjdbc_test_MyBridge *caller)
{
    return sizeof(int);
}
```

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// getINTValue
// Given a buffer, return the value as an int
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
long jdbc_test_MyBridge_getINTValue(
    struct Hjdbc_test_MyBridge *caller,
    HArrayOfByte *buf)
{
    // Cast our array of bytes to an integer pointer
    int* pInt = (int*) unhand (buf)->body;

    // Return the value
    return (long) *pInt;
}

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// callSomeFunction
// Call some function that takes a String and an int pointer as arguments
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
void jdbc_test_MyBridge_callSomeFunction(
    struct Hjdbc_test_MyBridge *caller,
    struct Hjava_lang_String *stringValue,
    HArrayOfByte *buf)
{
    // Cast the string into a char pointer
    char* pString = (char*) makeCString (stringValue);

    // Cast our array of bytes to an integer pointer
    int* pInt = (int*) unhand (buf)->body;

    // This fictitious function will print the string, then return the
    // length of the string in the int pointer.
    printf("String value=%s\n", pString);
    *pInt = strlen(pString);
}
```

Now, create a library (DLL or Shared Object) by compiling this module and linking it with the jdbc_test_MyDriver compiled object and the one required Java library, **javai.lib**. Here's the command line I used to build it for Win95/NT:

```
cl -DWIN32 mybridge.c jdbc_test_mybridge.c -FeMyBridge.dll -MD -LD
javai.lib
```

Now we can use our native bridge, as shown in Listing 10.11.

Listing 10.11 Implementing the bridge.

```
import jdbc.test.*;
import java.sql.*;

class Test {

    public static void main (String args[]) {

        MyBridge myBridge = null;
        boolean loaded = false;

        try {

            // Create a new bridge object. If it is unable to load our
            // native library, an SQLException will be thrown.
            myBridge = new MyBridge();
            loaded = true;
        }
        catch (SQLException ex) {
            System.out.println("SQLException: " + ex.getMessage());
        }

        // If the bridge was loaded, use the native methods
        if (loaded) {

            // Allocate storage for an int
            byte intValue[] = new byte[myBridge.getINTSize()];

            // Call the bridge to perform some function with a string,
            // returning a value in the int buffer.
            myBridge.callSomeFunction("Hello, World.", intValue);

            // Get the value out of the buffer.
            int n = myBridge.getINTValue(intValue);

            System.out.println("INT value=" + n);
        }
    }
}
```

Listing 10.11 produces the following output:

```
String value=Hello, World.
INT value=13
```

As you can see, using native methods is very straightforward. Developing a JDBC driver using a native bridge is a natural progression for existing database systems that provide a C API. The real power and ultimate solution, though, is to develop non-native JDBC drivers—those consisting of 100 percent Java code.

Implementing Interfaces

The JDBC API specification provides a series of *interfaces* that must be implemented by the JDBC driver developer. An interface declaration creates a new reference type consisting of constants and abstract methods. An interface cannot contain any implementations (that is, executable code). What does all of this mean? The JDBC API specification dictates the methods and method interfaces for the API, and a driver must fully implement these interfaces. A JDBC application makes method calls to the JDBC interface, not a specific driver. Because all JDBC drivers must implement the same interface, they are interchangeable.

There are a few rules that you must follow when implementing interfaces. First, you must implement the interface exactly as specified. This includes the name, return value, parameters, and **throws** clause. Secondly, you must be sure to implement all interfaces as **public** methods. Remember, this is the interface that other classes will see; if it isn't **public**, it can't be seen. Finally, all methods in the interface must be implemented. If you forget, the Java compiler will kindly remind you.

Take a look at Listing 10.12 for an example of how interfaces are used. The code defines an interface, implements the interface, and then uses the interface.

Listing 10.12 Working with interfaces.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// MyInterface.java
//
// Sample code to demonstrate the use of interfaces
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
package jdbc.test;

public interface MyInterface
{
```

```
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Define 3 methods in this interface
    //-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    void method1();
    int method2(int x);
    String method3(String y);
}
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// MyImplementation.java
//
// Sample code to demonstrate the use of interfaces
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

package jdbc.test;

public class MyImplementation
    implements jdbc.test.MyInterface
{
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Implement the 3 methods in the interface
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
       public void method1()
    {
    }

    public int method2(int x)
    {
       return addOne(x);
    }

    public String method3(String y)
    {
       return y;
    }

    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Note that you are free to add methods and attributes to this
    // new class that were not in the interface, but they cannot be
    // seen from the interface.
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    protected int addOne(int x)
    {
        return x + 1;
    }
```

```
}
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// TestInterface.java
//
// Sample code to demonstrate the use of interfaces
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
import jdbc.test.*;

class TestInterface {

    public static void main (String args[])
    {
        // Create a new MyImplementation object. We are assigning the
        // new object to a MyInterface variable, thus we will only be
        // able to use the interface methods.
        MyInterface myInterface = new MyImplementation();

        // Call the methods
        myInterface.method1();
        int x = myInterface.method2(1);
        String y = myInterface.method3("Hello, World.");

    }
}
```

As you can see, implementing interfaces is easy. We'll go into more detail with the major JDBC interfaces later in this chapter. But first, we need to cover some basic foundations that should be a part of every good JDBC driver.

Tracing

One detail that is often overlooked by software developers is providing a facility to enable debugging. The JDBC API does provide methods to enable and disable tracing, but it is ultimately up to the driver developer to provide tracing information in the driver. It becomes even more critical to provide a detailed level of tracing when you consider the possible wide-spread distribution of your driver. People from all over the world may be using your software, and they will expect a certain level of support if problems arise. For this reason, I consider it a must to trace all of the JDBC API method calls (so that a problem can be re-created using the output from a trace).

Turning On Tracing

The **DriverManager** provides a method to set the tracing **PrintStream** to be used for all of the drivers; not only those that are currently active, but any drivers that are subsequently loaded. Note that if two applications are using JDBC, and both have turned tracing on, the **PrintStream** that is set last will be shared by both applications. The following code snippet shows how to turn tracing on, sending any trace messages to a local file:

```
try {
    // Create a new OuputStream using a file. This may fail if the
    // calling application/applet does not have the proper security
    // to write to a local disk.
    java.io.OutputStream outFile = new
    java.io.FileOutputStream("jdbc.out");

    // Create a PrintStream object using our newly created OuputStream
    // object. The second parameter indicates to flush all output with
    // each write. This ensures that all trace information gets written
    // into the file.
    java.io.PrintStream outStream = new java.io.PrintStream(outFile,
    true);

    // Enable the JDBC tracing, using the PrintStream
    DriverManager.setLogStream(outStream);
}
catch (Exception ex) {
    // Something failed during enabling JDBC tracing. Notify the
    // application that tracing is not available.
    .
    .
    .
}
```

Using this code, a new file named jdbc.out will be created (if an existing file already exists, it will be overwritten), and any tracing information will be saved in the file.

Writing Tracing Information

The **DriverManager** also provides a method to write information to the tracing **OutputStream**. The **println** method will first check to ensure that a trace **OutputStream** has been registered, and if so, the **println** method of the **OutputStream** will be called. Here's an example of writing trace information:

```
// Send some information to the JDBC trace OutputStream
String a = "The quick brown fox ";
String b = "jumped over the ";
String c = "lazy dog";

DriverManager.println("Trace=" + a + b + c);
```

In this example, a **String** message of "Trace=The quick brown fox jumped over the lazy dog" will be constructed, the message will be provided as a parameter to the **DriverManager.println** method, and the message will be written to the **OutputStream** being used for tracing (if one has been registered).

Some of the JDBC components are also nice enough to provide tracing information. The **DriverManager** object traces most of its method calls. **SQLException** also sends trace information whenever an exception is thrown. If you were to use the previous code example and enable tracing to a file, the following example output will be created when attempting to connect to the SimpleText driver:

```
DriverManager.initialize: jdbc.drivers = null
JDBC DriverManager initialized
registerDriver:
driver[className=jdbc.SimpleText.SimpleTextDriver,context=null,
jdbc.SimpleText.SimpleTextDriver@1393860]
DriverManager.getConnection("jdbc:SimpleText")
trying
driver[className=jdbc.SimpleText.SimpleTextDriver,context=null,
jdbc.SimpleText.SimpleTextDriver@1393860]
driver[className=jdbc.SimpleText.SimpleTextDriver,context=null,j
dbc.SimpleText.SimpleTextDriver@1393860]
```

Checking For Tracing

I have found it quite useful for both the application and the driver to be able to test for the presence of a tracing **PrintStream**. The JDBC API provides us with a method to determine if tracing is enabled, as shown here:

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// traceOn
// Returns true if tracing (logging) is currently enabled
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
public static boolean traceOn()
{
    // If the DriverManager log stream is not null, tracing
    // must be currently enabled.
    return (DriverManager.getLogStream() != null);
}
```

From an application, you can use this method to check if tracing has been previously enabled before blindly setting it:

```
// Before setting tracing on, check to make sure that tracing is not
// already turned on. If it is, notify the application.
if (traceOn()) {
    // Issue a warning that tracing is already enabled
   .
   .
   .
}
```

From the driver, I use this method to check for tracing before attempting to send information to the **PrintStream**. In the example where we traced the message text of "Trace=The quick brown fox jumped over the lazy dog," a lot had to happen before the message was sent to the **DriverManager.println** method. All of the given **String** objects had to be concatenated, and a new **String** had to be constructed. That's a lot of overhead to go through before even making the **println** call, especially if tracing is not enabled (which will probably be the majority of the time). So, for performance reasons, I prefer to ensure that tracing has been enabled before assembling my trace message:

```
// Send some information to the JDBC trace OutputStream
String a = "The quick brown fox ";
String b = "jumped over the ";
String c = "lazy dog";

// Make sure tracing has been enabled
if (traceOn()) {
    DriverManager.println("Trace=" + a + b + c);
}
```

Data Coercion

At the heart of every JDBC driver is data. That is the whole purpose of the driver: providing data. Not only providing it, but providing it in a requested

format. This is what data coercion is all about—converting data from one format to another. As Figure 10.1 shows, JDBC specifies the necessary conversions.

In order to provide reliable data coercion, a data wrapper class should be used. This class contains a data value in some known format and provides methods to convert it to a specific type. As an example, I have included the **CommonValue** class from the SimpleText driver in Listing 10.13. This class has several overloaded constructors that accept different types of data values. The data value is stored within the class, along with the type of data (String, Integer, etc.). A series of methods are then provided to get the data

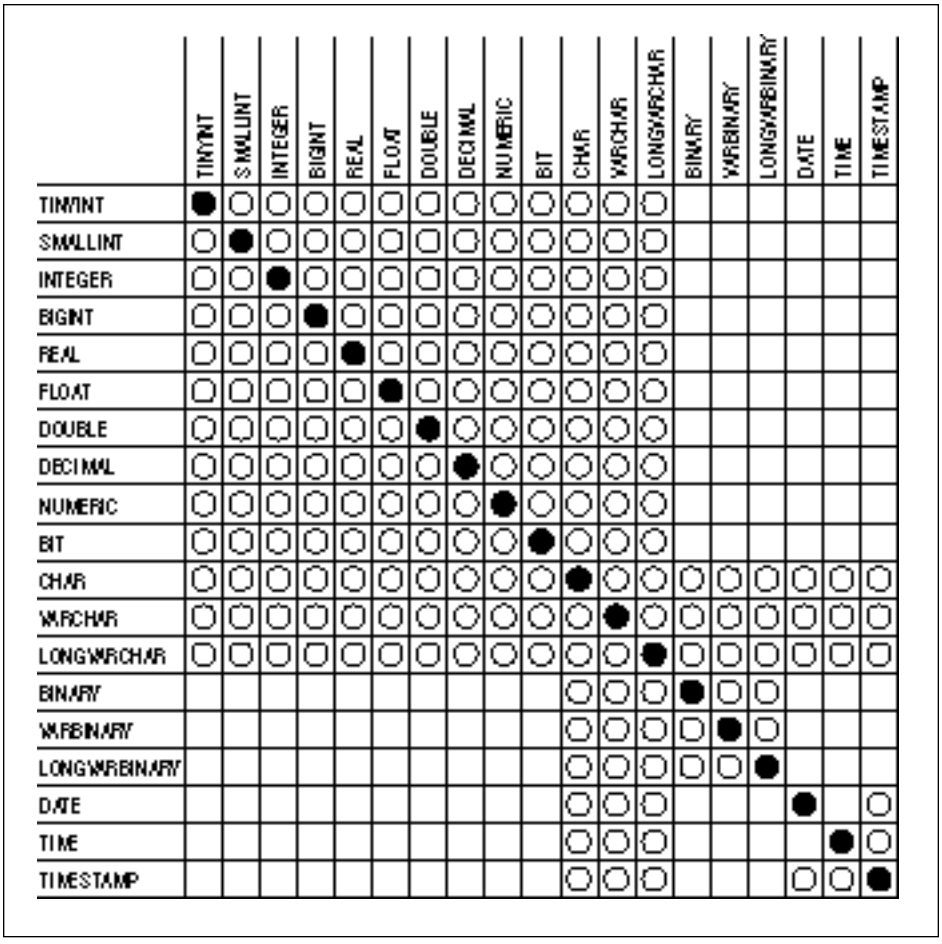


Figure 10.1 JDBC data conversion table.

in different formats. This class greatly reduces the burden of the JDBC driver developer, and can serve as a fundamental class for any number of drivers.

Listing 10.13 The CommonValue class.

```
package jdbc.SimpleText;

import java.sql.*;

public class CommonValue
    extends         Object
{
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Constructors
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    public CommonValue()
    {
        data = null;
    }

    public CommonValue(String s)
    {
        data = (Object) s;
        internalType = Types.VARCHAR;
    }

    public CommonValue(int i)
    {
        data = (Object) new Integer(i);
        internalType = Types.INTEGER;
    }

    public CommonValue(Integer i)
    {
        data = (Object) i;
        internalType = Types.INTEGER;
    }

    public CommonValue(byte b[])
    {
        data = (Object) b;
        internalType = Types.VARBINARY;
    }
```

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// isNull
// returns true if the value is null
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
public boolean isNull()
{
    return (data == null);
}

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// getMethods
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

// Attempt to convert the data into a String. All data types
// should be able to be converted.
public String getString()
    throws SQLException
{
    String s;

    // A null value always returns null
    if (data == null) {
        return null;
    }

    switch(internalType) {

    case Types.VARCHAR:
        s = (String) data;
        break;

    case Types.INTEGER:
        s = ((Integer) data).toString();
        break;

    case Types.VARBINARY:
        {
            // Convert a byte array into a String of hex digits
            byte b[] = (byte[]) data;
            int len = b.length;
            String digits = "0123456789ABCDEF";
            char c[] = new char[len * 2];

            for (int i = 0; i < len; i++) {
```

```
                c[i * 2] = digits.charAt((b[i] >> 4) & 0x0F);
                c[(i * 2) + 1] = digits.charAt(b[i] & 0x0F);
            }
            s = new String(c);
        }
        break;

    default:
        throw new SQLException("Unable to convert data type to
          String: " +
                           internalType);
    }

    return s;
}

// Attempt to convert the data into an int
public int getInt()
    throws SQLException
{
    int i = 0;

    // A null value always returns zero
    if (data == null) {
        return 0;
    }

    switch(internalType) {

    case Types.VARCHAR:
        i = (Integer.valueOf((String) data)).intValue();
        break;

    case Types.INTEGER:
        i = ((Integer) data).intValue();
        break;

    default:
        throw new SQLException("Unable to convert data type to
          String: " +
                           internalType);
    }

    return i;
}
```

```
// Attempt to convert the data into a byte array
public byte[] getBytes()
    throws SQLException
{
    byte b[] = null;

    // A null value always returns null
    if (data == null) {
        return null;
    }

    switch(internalType) {

    case Types.VARCHAR:
        {
            // Convert the String into a byte array. The String must
            // contain an even number of hex digits.
            String s = ((String) data).toUpperCase();
            String digits = "0123456789ABCDEF";
            int len = s.length();
            int index;

            if ((len % 2) != 0) {
                throw new SQLException(
                        "Data must have an even number of hex
                          digits");
            }

            b = new byte[len / 2];

            for (int i = 0; i < (len / 2); i++) {
                index = digits.indexOf(s.charAt(i * 2));

                if (index < 0) {
                    throw new SQLException("Invalid hex digit");
                }

                b[i] = (byte) (index << 4);
                index = digits.indexOf(s.charAt((i * 2) + 1));

                if (index < 0) {
                    throw new SQLException("Invalid hex digit");
                }
                b[i] += (byte) index;
            }
```

```
            }
            break;

        case Types.VARBINARY:
            b = (byte[]) data;
            break;

        default:
            throw new SQLException("Unable to convert data type to
              byte[]: " +
                                internalType);
        }
        return b;
    }

    protected Object data;
    protected int internalType;
}
```

Note that the SimpleText driver supports only character, integer, and binary data; thus, **CommonValue** only accepts these data types, and only attempts to convert data to these same types. A more robust driver would need to further implement this class to include more (if not all) data types.

Escape Clauses

Another consideration when implementing a JDBC driver is processing escape clauses. Escape clauses are used as extensions to SQL and provide a method to perform DBMS-specific extensions, which are interoperable among DBMSes. The JDBC driver must accept escape clauses and expand them into the native DBMS format before processing the SQL statement. While this sounds simple enough on the surface, this process may turn out to be an enormous task. If you are developing a driver that uses an existing DBMS, and the JDBC driver simply passes SQL statements to the DBMS, you may have to develop a parser to scan for escape clauses.

The following types of SQL extensions are defined:

- Date, time, and timestamp data
- Scalar functions such as numeric, string, and data type conversion

- LIKE predicate escape characters
- Outer joins
- Procedures

The JDBC specification does not directly address escape clauses; they are inherited from the ODBC specification. The syntax defined by ODBC uses the escape clause provided by the X/OPEN and SQL Access Group SQL CAE specification (1992). The general syntax for an escape clause is:

```
{escape}
```

We'll cover the specific syntax for each type of escape clause in the following sections.

Date, Time, And Timestamp

The **date**, **time**, and **timestamp** escape clauses allow an application to specify date, time, and timestamp data in a uniform manner, without concern to the native DBMS format (for which the JDBC driver is responsible). The syntax for each (respectively) is

```
{d 'value'}
{t 'value'}
{ts 'value'}
```

where **d** indicates *value* is a date in the format yyyy-mm-dd, **t** indicates *value* is a time in the format hh:mm:ss, and **ts** indicates *value* is a timestamp in the format yyyy-mm-dd hh:mm:ss[.f...]. The following SQL statements illustrate the use of each:

```
UPDATE EMPLOYEE SET HIREDATE={d '1992-04-01'}
UPDATE EMPLOYEE SET LAST_IN={ts '1996-07-03 08:00:00'}
UPDATE EMPLOYEE SET BREAK_DUE={t '10:00:00'}
```

Scalar Functions

The five types of scalar functions—string, numeric, time and date, system, and data type conversion—all use the syntax:

```
{fn scalar-function}
```

To determine what type of string functions a JDBC driver supports, an application can use the DatabaseMetaData method **getStringFunctions**. This method returns a comma-separated list of string functions, possibly containing ASCII, CHAR, CONCAT, DIFFERENCE, INSERT, LCASE, LEFT, LENGTH, LOCATE, LTRIM, REPEAT, REPLACE, RIGHT, RTRIM, SOUNDEX, SPACE, SUBSTRING, and/or UCASE.

To determine what type of numeric functions a JDBC driver supports, an application can use the DatabaseMetaData method **getNumericFunctions**. This method returns a comma-separated list of numeric functions, possibly containing ABS, ACOS, ASIN, ATAN, ATAN2, CEILING, COS, COT, DEGREES, EXP, FLOOR, LOG, LOG10, MOD, PI, POWER, RADIANS, RAND, ROUND, SIGN, SIN, SQRT, TAN, and/or TRUNCATE.

To determine what type of system functions a JDBC driver supports, an application can use the **DatabaseMetaData** method **getSystemFunctions**. This method returns a comma-separated list of system functions, possibly containing DATABASE, IFNULL, and/or USER.

To determine what type of time and date functions a JDBC driver supports, an application can use the **DatabaseMetaData** method **getTimeDateFunctions**. This method returns a comma-separated list of time and date functions, possibly containing CURDATE, CURTIME, DAYNAME, DAYOFMONTH, DAYOFWEEK, DAYOFYEAR, HOUR, MINUTE, MONTH, MONTHNAME, NOW, QUARTER, SECOND, TIMESTAMPADD, TIMESTAMPDIFF, WEEK, and/or YEAR.

To determine what type of explicit data type conversions a JDBC driver supports, an application can use the **DatabaseMetaData** method **supportsConvert**. This method has two parameters: a *from* SQL data type and a *to* SQL data type. If the explicit data conversion between the two SQL types is supported, the method returns true. The syntax for the CONVERT function is

```
{fn CONVERT(value, data_type)}
```

where *value* is a column name, the result of another scalar function, or a literal, and *data_type* is one of the JDBC SQL types listed in the **Types** class.

LIKE Predicate Escape Characters

In a LIKE predicate, the "%" (percent character) matches zero or more of any character, and the "_" (underscore character) matches any one character. In some instances, an SQL query may have the need to search for one of these special matching characters. In such cases, you can use the "%" and "_" characters as literals in a LIKE predicate by preceding them with an escape character. The **DatabaseMetaData** method **getSearchStringEscape** returns the default escape character (which for most DBMSes will be the backslash character "\"). To override the escape character, use the following syntax:

```
{escape 'escape-character'}
```

The following SQL statement uses the LIKE predicate escape clause to search for any columns that start with the "%" character:

```
SELECT * FROM EMPLOYEE WHERE NAME LIKE '\%' {escape '\'}
```

Outer Joins

JDBC supports the ANSI SQL-92 LEFT OUTER JOIN syntax. The escape clause syntax is

```
{oj outer-join}
```

where *outer-join* is the table-reference LEFT OUTER JOIN {table-reference | outer-join} ON search-condition.

Procedures

A JDBC application can call a procedure in place of an SQL statement. The escape clause used for calling a procedure is

```
{[?=] call procedure-name[(param[, param]...)]}
```

where *procedure-name* specifies the name of a procedure stored on the data source, and *param* specifies procedure parameters. A procedure can have zero or more parameters, and may return a value.

The JDBC Interfaces

Now let's take a look at each of the JDBC interfaces, which are shown in Figure 10.2. We'll go over the major aspects of each interface and use code examples from our SimpleText project whenever applicable. You should understand the JDBC API specification before attempting to create a JDBC driver; this section is meant to enhance the specification, not to replace it.

Driver

The **Driver** class is the entry point for all JDBC drivers. From here, a connection to the database can be made in order to perform work. This class is intentionally very small; the intent is that JDBC drivers can be pre-registered with the system, enabling the **DriverManager** to select an appropriate driver given only a *URL* (Universal Resource Locator). The only way to

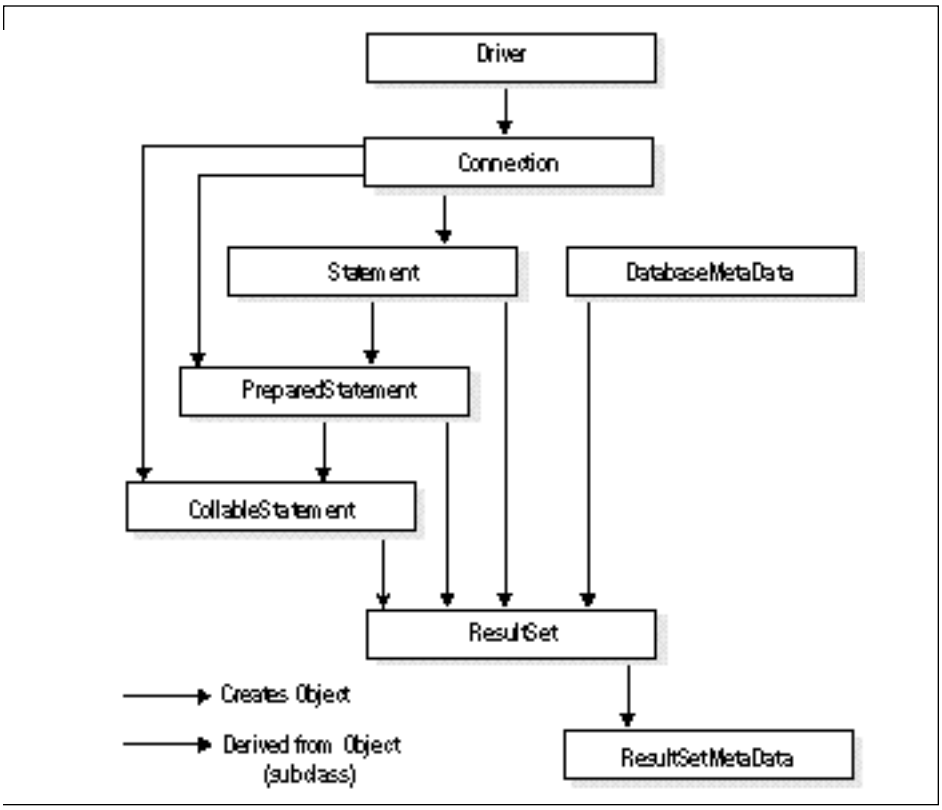


Figure 10.2 The JDBC interfaces.

determine which driver can service the given URL is to load the **Driver** class and let each driver respond via the **acceptsURL** method. To keep the amount of time required to find an appropriate driver to a minimum, each **Driver** class should be as small as possible so it can be loaded quickly.

Register Thyself

The very first thing that a driver should do is register itself with the **DriverManager**. The reason is simple: You need to tell the **DriverManager** that you exist; otherwise you may not be loaded. The following code illustrates one way of loading a JDBC driver:

```
java.sql.Driver d = (java.sql.Driver)
    Class.forName ("jdbc.SimpleText.SimpleTextDriver").newInstance();

Connection con = DriverManager.getConnection("jdbc:SimpleText", "", "");
```

The class loader will create a new instance of the SimpleText JDBC driver. The application then asks the **DriverManager** to create a connection using the given URL. If the SimpleText driver does not register itself, the **DriverManager** will not attempt to load it, which will result in a nasty "No capable driver" error.

The best place to register a driver is in the **Driver** constructor:

```
public SimpleTextDriver()
    throws SQLException
{
    // Attempt to register this driver with the JDBC DriverManager.
    // If it fails, an exception will be thrown.
    DriverManager.registerDriver(this);
}
```

URL Processing

As I mentioned a moment ago, the **acceptsURL** method informs the **DriverManager** whether a given URL is supported by the driver. The general format for a JDBC URL is

```
jdbc:subprotocol:subname
```

where *subprotocol* is the particular database connectivity mechanism supported (note that this mechanism may be supported by multiple drivers)

and the *subname* is defined by the JDBC driver. For example, the format for the JDBC-ODBC Bridge URL is:

```
jdbc:odbc:data source name
```

Thus, if an application requests a JDBC driver to service the URL of

```
jdbc:odbc:foobar
```

the only driver that will respond that the URL is supported is the JDBC-ODBC Bridge; all others will ignore the request.

Listing 10.14 shows the **acceptsURL** method for the SimpleText driver. The SimpleText driver will accept the following URL syntax:

```
jdbc:SimpleText
```

Note that no subname is required; if a subname is provided, it will be ignored.

Listing 10.14 The acceptsURL method.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// acceptsURL - JDBC API
//
// Returns true if the driver thinks that it can open a connection
// to the given URL. Typically, drivers will return true if they
// understand the subprotocol specified in the URL, and false if
// they don't.
//
//    url        The URL of the database.
//
// Returns true if this driver can connect to the given URL.
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
public boolean acceptsURL(
    String url)
    throws SQLException
{
    if (traceOn()) {
        trace("@acceptsURL (url=" + url + ")");
    }

    boolean rc = false;
```

```
    // Get the subname from the url. If the url is not valid for
    // this driver, a null will be returned.
    if (getSubname(url) != null) {
        rc = true;
    }

    if (traceOn()) {
        trace(" " + rc);
    }
    return rc;
}

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// getSubname
// Given a URL, return the subname. Returns null if the protocol is
// not "jdbc" or the subprotocol is not "simpletext."
//-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
public String getSubname(
    String url)
{
    String subname = null;
    String protocol = "JDBC";
    String subProtocol = "SIMPLETEXT";

    // Convert to uppercase and trim all leading and trailing
    // blanks.
    url = (url.toUpperCase()).trim();

    // Make sure the protocol is jdbc:
    if (url.startsWith(protocol)) {

        // Strip off the protocol
        url = url.substring (protocol.length());

        // Look for the colon
        if (url.startsWith(":")) {
            url = url.substring(1);

            // Check the subprotocol
            if (url.startsWith(subProtocol)) {

                // Strip off the subprotocol, leaving the subname
                url = url.substring(subProtocol.length());
```

```
                // Look for the colon that separates the subname
                // from the subprotocol (or the fact that there
                // is no subprotocol at all).
                if (url.startsWith(":")) {
                    subname = url.substring(subProtocol.length());
                }
                else if (url.length() == 0) {
                    subname = "";
                }
            }
        }
    }
    return subname;
}
```

Driver Properties

Connecting to a JDBC driver with only a URL specification is great, but the vast majority of the time, a driver will require additional information in order to properly connect to a database. The JDBC specification has addressed this issue with the **getPropertyInfo** method. Once a **Driver** has been instantiated, an application can use this method to find out what required and optional properties can be used to connect to the database. You may be tempted to require the application to embed properties within the URL subname, but by returning them from the **getPropertyInfo** method, you can identify the properties at runtime, giving a much more robust solution. Listing 10.15 shows an application that loads the SimpleText driver and gets the property information.

Listing 10.15 Using the getPropertyInfo method to identify properties at runtime.

```
import java.sql.*;

class PropertyTest {

    public static void main(String args[])
    {
        try {

            // Quick way to create a driver object
            java.sql.Driver d = new jdbc.SimpleText.SimpleTextDriver();
```

```
            String url = "jdbc:SimpleText";

            // Make sure we have the proper URL
            if (!d.acceptsURL(url)) {
                throw new SQLException("Unknown URL: " + url);
            }

            // Setup a Properties object. This should contain an entry
            // for all known properties to this point. Properties that
            // have already been specified in the Properties object will
            // not be returned by getPropertyInfo.
            java.util.Properties props = new java.util.Properties();

            // Get the property information
            DriverPropertyInfo info[] = d.getPropertyInfo(url, props);

            // Just dump them out
            System.out.println("Number of properties: " + info.length);

            for (int i=0; i < info.length; i++) {
                System.out.println("\nProperty " + (i + 1));
                System.out.println("Name:        " + info[i].name);
                System.out.println("Description: " +
                  info[i].description);
                System.out.println("Required:    " + info[i].required);
                System.out.println("Value:       " + info[i].value);
                System.out.println("Choices:     " + info[i].choices);
            }

        }
        catch (SQLException ex) {
            System.out.println ("\nSQLException(s) caught\n");

            // Remember that SQLExceptions may be chained together
            while (ex != null) {
                System.out.println("SQLState: " + ex.getSQLState());
                System.out.println("Message:  " + ex.getMessage());
                System.out.println ("");
                ex = ex.getNextException ();
            }
        }
    }
}
```

Listing 10.15 produces the following output:

```
Number of properties: 1

Property 1
Name:        Directory
Description: Initial text file directory
Required:    false
Value:       null
Choices:     null
```

It doesn't take a lot of imagination to envision an application or applet that gathers the property information and prompts the user in order to connect to the database. The actual code to implement the **getPropertyInfo** method for the SimpleText driver is very simple, as shown in Listing 10.16.

Listing 10.16 Implementing the getPropertyInfo method.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// getPropertyInfo - JDBC API
//
// The getPropertyInfo method is intended to allow a generic GUI tool to
// discover what properties it should prompt a human for in order to get
// enough information to connect to a database. Note that depending on
// the values the human has supplied so far, additional values may become
// necessary, so it may be necessary to iterate though several calls.
// to getPropertyInfo.
//
//    url        The URL of the database to connect to.
//
//    info    A proposed list of tag/value pairs that will be sent on
//            connect open.
//
// Returns an array of DriverPropertyInfo objects describing possible
//            properties. This array may be an empty array if no
//            properties are required.
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

public DriverPropertyInfo[] getPropertyInfo(
    String url,
    java.util.Properties info)
    throws SQLException
{
    DriverPropertyInfo prop[];
```

```
    // Only one property required for the SimpleText driver, the
    // directory. Check the property list coming in. If the
    // directory is specified, return an empty list.
    if (info.getProperty("Directory") == null) {

        // Setup the DriverPropertyInfo entry
        prop = new DriverPropertyInfo[1];
        prop[0] = new DriverPropertyInfo("Directory", null);
        prop[0].description = "Initial text file directory";
        prop[0].required = false;

    }
    else {
        // Create an empty list
        prop = new DriverPropertyInfo[0];
    }

    return prop;
}
```

Let's Get Connected

Now that we can identify a driver to provide services for a given URL and get a list of the required and optional parameters necessary, it's time to establish a connection to the database. The **connect** method does just that, as shown in Listing 10.17, by taking a URL and connection property list and attempting to make a connection to the database. The first thing that **connect** should do is verify the URL (by making a call to **acceptsURL**). If the URL is not supported by the driver, a null value will be returned. This is the only reason that a null value should be returned. Any other errors during the **connect** should throw an **SQLException**.

Listing 10.17 Connecting to the database.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// connect - JDBC API
//
// Try to make a database connection to the given URL.
// The driver should return "null" if it realizes it is the wrong kind
// of driver to connect to the given URL. This will be common, as when
// the JDBC driver manager is asked to connect to a given URL, it passes
// the URL to each loaded driver in turn.
//
// The driver should raise an SQLException if it is the right
```

```
// driver to connect to the given URL, but has trouble connecting to
// the database.
//
// The java.util.Properties argument can be used to pass arbitrary
// string tag/value pairs as connection arguments.
// Normally, at least "user" and "password" properties should be
// included in the Properties.
//
//    url    The URL of the database to connect to.
//
//    info   a list of arbitrary string tag/value pairs as
//           connection arguments; normally, at least a "user" and
//           "password" property should be included.
//
// Returns a Connection to the URL.
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
public Connection connect(
    String url,
    java.util.Properties info)
    throws SQLException
{
    if (traceOn()) {
        trace("@connect (url=" + url + ")");
    }

    // Ensure that we can understand the given URL
    if (!acceptsURL(url)) {
        return null;
    }

    // For typical JDBC drivers, it would be appropriate to check
    // for a secure environment before connecting, and deny access
    // to the driver if it is deemed to be unsecure. For the
    // SimpleText driver, if the environment is not secure, we will
    // turn it into a read-only driver.

    // Create a new SimpleTextConnection object
    SimpleTextConnection con = new SimpleTextConnection();

    // Initialize the new object. This is where all of the
    // connection work is done.
    con.initialize(this, info);

    return con;
}
```

As you can see, there isn't a lot going on here for the SimpleText driver; remember that we need to keep the size of the **Driver** class implementation as small as possible. To aid in this, all of the code required to perform the database connection resides in the **Connection** class, which we'll discuss next.

Connection

The **Connection** class represents a session with the data source. From here, you can create **Statement** objects to execute SQL statements and gather database statistics. Depending upon the database that you are using, multiple connections may be allowed for each driver.

For the SimpleText driver, we don't need to do anything more than actually connect to the database. In fact, there really isn't a database at all—just a bunch of text files. For typical database drivers, some type of connection context will be established, and default information will be set and gathered. During the SimpleText connection initialization, all that we need to do is check for a read-only condition (which can only occur within untrusted applets) and any properties that are supplied by the application, as shown in Listing 10.18.

Listing 10.18 SimpleText connection initialization.

```
public void initialize(
    Driver driver,
    java.util.Properties info)
    throws SQLException
{
    // Save the owning driver object
    ownerDriver = driver;

    // Get the security manager and see if we can write to a file.
    // If no security manager is present, assume that we are a trusted
    // application and have read/write privileges.
    canWrite = false;

    SecurityManager securityManager = System.getSecurityManager ();

    if (securityManager != null) {
        try {
            // Use some arbitrary file to check for file write privileges
            securityManager.checkWrite ("SimpleText_Foo");
```

```
            // Flag is set if no exception is thrown
            canWrite = true;
        }

        // If we can't write, an exception is thrown. We'll catch
        // it and do nothing.
        catch (SecurityException ex) {
        }
    }
    else {
        canWrite = true;
    }

    // Set our initial read-only flag
    setReadOnly(!canWrite);

    // Get the directory. It will either be supplied in the property
    // list, or we'll use our current default.
    String s = info.getProperty("Directory");

    if (s == null) {
        s = System.getProperty("user.dir");
    }

    setCatalog(s);
}
```

Creating Statements

From the **Connection** object, an application can create three types of **Statement** objects. The base **Statement** object is used for executing SQL statements directly. The **PreparedStatement** object (which extends **Statement**) is used for pre-compiling SQL statements that may contain input parameters. The **CallableStatement** object (which extends **PreparedStatement**) is used to execute stored procedures that may contain both input and output parameters.

For the SimpleText driver, the **createStatement** method does nothing more than create a new **Statement** object. For most database systems, some type of statement context, or handle, will be created. One thing to note whenever an object is created in a JDBC driver: Save a reference to the owning object because you will need to obtain information (such as the connection context from within a **Statement** object) from the owning object.

Consider the **createStatement** method within the **Connection** class:

```
public Statement createStatement()
    throws SQLException
{
    if (traceOn()) {
        trace("Creating new SimpleTextStatement");
    }

    // Create a new Statement object
    SimpleTextStatement stmt = new SimpleTextStatement();

    // Initialize the statement
    stmt.initialize(this);

    return stmt;
}
```

Now consider the corresponding **initialize** method in the **Statement** class:

```
public void initialize(
    SimpleTextConnection con)
    throws SQLException
{
    // Save the owning connection object
    ownerConnection = con;
}
```

Which module will you compile first? You can't compile the **Connection** class until the **Statement** class has been compiled, and you can't compile the **Statement** class until the **Connection** class has been compiled. This is a circular dependency. Of course, the Java compiler does allow multiple files to be compiled at once, but some build environments do not support circular dependency. I have solved this problem in the SimpleText driver by defining some simple interface classes. In this way, the **Statement** class knows only about the general interface of the **Connection** class; the implementation of the interface does not need to be present. Our modified **initialize** method looks like this:

```
public void initialize(
    SimpleTextIConnection con)
    throws SQLException
```

```
{
    // Save the owning connection object
    ownerConnection = con;
}
```

Note that the only difference is the introduction of a new class, **SimpleTextIConnection**, which replaces **SimpleTextConnection**. I have chosen to preface the JDBC class name with an "I" to signify an interface. Here's the interface class:

```
public interface SimpleTextIConnection
    extends java.sql.Connection
{
    String[] parseSQL(String sql);
    Hashtable getTables(String directory, String table);
    Hashtable getColumns(String directory, String table);
    String getDirectory(String directory);
}
```

Note that our interface class extends the JDBC class, and our **Connection** class implements this new interface. This allows us to compile the interface first, then the **Statement**, followed by the **Connection**. Say good-bye to your circular dependency woes.

Now, back to the **Statement** objects. The **prepareStatement** and **prepareCall** methods of the **Connection** object both require an SQL statement to be provided. This SQL statement should be pre-compiled and stored with the **Statement** object. If any errors are present in the SQL statement, an exception should be raised, and the **Statement** object should not be created.

Tell Me About Yourself

One of the most powerful aspects of the JDBC specification (which was inherited from X/Open) is the ability for introspection. This is the process of asking a driver for information about what is supported, how it behaves, and what type of information exists in the database. The **getMetaData** method creates a **DatabaseMetaData** object which provides us with this wealth of information.

DatabaseMetaData

At over 130 methods, the **DatabaseMetaData** class is by far the largest. It supplies information about what is supported and how things are supported. It also supplies catalog information such as listing tables, columns, indexes, procedures, and so on. Because the JDBC API specification does an adequate job of explaining the methods contained in this class, and most of them are quite straightforward, we'll just take a look at how the SimpleText driver implements the **getTables** catalog method. But first, let's review the basic steps needed to implement each of the catalog methods (that is, those methods that return a **ResultSet**):

1. Create the result columns, which includes the column name, type, and other information about each of the columns. You should perform this step regardless of whether the database supports a given catalog function (such as stored procedures). I believe that it is much better to return an empty result set with only the column information than to raise an exception indicating that the database does not support the function. The JDBC specification does not currently address this issue, so it is open for interpretation.
2. Retrieve the catalog information from the database.
3. Perform any filtering necessary. The application may have specified the return of only a subset of the catalog information. You may need to filter the information in the JDBC driver if the database system doesn't.
4. Sort the result data per the JDBC API specification. If you are lucky, the database you are using will sort the data in the proper sequence. Most likely, it will not. In this case, you will need to ensure that the data is returned in the proper order.
5. Return a **ResultSet** containing the requested information.

The SimpleText **getTables** method will return a list of all of the text files in the catalog (directory) given. If no catalog is supplied, the default directory is used. Note that the SimpleText driver does not perform all of the steps shown previously; it does not provide any filtering, nor does it sort the data in the proper sequence. You are more than welcome to

add this functionality. In fact, I encourage it. One note about column information: I prefer to use a **Hashtable** containing the column number as the key, and a class containing all of the information about the column as the data value. So, for all **ResultSets** that are generated, I create a **Hashtable** of column information that is then used by the **ResultSet** object and the **ResultSetMetaData** object to describe each column. Listing 10.19 shows the **SimpleTextColumn** class that is used to hold this information for each column.

Listing 10.19 The SimpleTextColumn class.

```
package jdbc.SimpleText;

public class SimpleTextColumn
    extends        Object
{
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Constructor
    //-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    public SimpleTextColumn(
        String name,
        int type,
        int precision)
    {
        this.name = name;
        this.type = type;
        this.precision = precision;
    }

    public SimpleTextColumn(
        String name,
        int type)
    {
        this.name = name;
        this.type = type;
        this.precision = 0;
    }

    public SimpleTextColumn(
        String name)
    {
        this.name = name;
        this.type = 0;
```

```
        this.precision = 0;
    }

    public String name;
    public int type;
    public int precision;
    public boolean searchable;
    public int colNo;
    public int displaySize;
    public String typeName;
}
```

Note that I have used several constructors to set up various default information, and that all of the attributes are **public**. To follow object-oriented design, I should have provided a **get** and **set** method to encapsulate each attribute, but I chose to let each consumer of this object access them directly. Listing 10.20 shows the code for the **getTables** method.

Listing 10.20 The getTables method.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// getTables - JDBC API
// Get a description of tables available in a catalog
//
// Only table descriptions matching the catalog, schema, table
// name and type criteria are returned. They are ordered by
// TABLE_TYPE, TABLE_SCHEM, and TABLE_NAME.
//
// Each table description has the following columns:
//
//    (1) TABLE_CAT     String => table catalog (may be null)
//    (2) TABLE_SCHEM   String => table schema (may be null)
//    (3) TABLE_NAME    String => table name
//    (4) TABLE_TYPE    String => table type
//            Typical types are "TABLE", "VIEW", "SYSTEM TABLE",
//            "GLOBAL TEMPORARY", "LOCAL TEMPORARY", "ALIAS", "SYNONYM"
//    (5) REMARKS       String => explanatory comment on the table
//
// Note: Some databases may not return information for
// all tables.
//
//    catalog            a catalog name; "" retrieves those without a
//                       catalog.
//    schemaPattern    a schema name pattern; "" retrieves those
//                     without a schema.
```

```
//    tableNamePattern    a table name pattern.
//    types            a list of table types to include; null returns all
//                     types.
//
// Returns a ResultSet. Each row is a table description.
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
public ResultSet getTables(
    String catalog,
    String schemaPattern,
    String tableNamePattern,
    String types[])
    throws SQLException
{
    if (traceOn()) {
        trace("@getTables(" + catalog + ", " + schemaPattern +
                ", " + tableNamePattern + ")");
    }

    // Create a statement object
    SimpleTextStatement stmt =
                (SimpleTextStatement) ownerConnection.createStatement();

    // Create a Hashtable for all of the columns
    Hashtable columns = new Hashtable();

    add(columns, 1, "TABLE_CAT", Types.VARCHAR);
    add(columns, 2, "TABLE_SCHEM", Types.VARCHAR);
    add(columns, 3, "TABLE_NAME", Types.VARCHAR);
    add(columns, 4, "TABLE_TYPE", Types.VARCHAR);
    add(columns, 5, "REMARKS", Types.VARCHAR);

    // Create an empty Hashtable for the rows
    Hashtable rows = new Hashtable();

    // If any of the parameters will return an empty result set, do so
    boolean willBeEmpty = false;

    // If table types are specified, make sure that 'TABLE' is
    // included. If not, no rows will be returned.

    if (types != null) {
        willBeEmpty = true;
        for (int ii = 0; ii < types.length; ii++) {
            if (types[ii].equalsIgnoreCase("TABLE")) {
                willBeEmpty = false;
```

```
                break;
            }
        }
    }

    if (!willBeEmpty) {

        // Get a Hashtable with all tables
        Hashtable tables = ownerConnection.getTables(
                    ownerConnection.getDirectory(catalog),
                      tableNamePattern);

        Hashtable singleRow;
        SimpleTextTable table;

        // Create a row for each table in the Hashtable
        for (int i = 0; i < tables.size(); i++) {
            table = (SimpleTextTable) tables.get(new Integer(i));

            // Create a new Hashtable for a single row
            singleRow = new Hashtable();

            // Build the row
            singleRow.put(new Integer(1), new CommonValue(table.dir));
            singleRow.put(new Integer(3), new CommonValue(table.name));
            singleRow.put(new Integer(4), new CommonValue("TABLE"));

            // Add it to the row list
            rows.put(new Integer(i + 1), singleRow);
        }
    }

    // Create the ResultSet object and return it
    SimpleTextResultSet rs = new SimpleTextResultSet();

    rs.initialize(stmt, columns, rows);

    return rs;
}
```

Let's take a closer look at what's going on here. The first thing we do is create a **Statement** object to "fake out" the **ResultSet** object that we will be creating to return back to the application. The **ResultSet** object is dependent upon a **Statement** object, so we'll give it one. The next thing we do is

create all of the column information. Note that all of the required columns are given in the JDBC API specification. The **add** method simply adds a **SimpleTextColumn** object to the **Hashtable** of columns:

```
protected void add(
    Hashtable h,
    int col,
    String name,
    int type)
{
    h.put(new Integer(col), new SimpleTextColumn(name,type));
}
```

Next, we create another **Hashtable** to hold all of the data for all of the catalog rows. The **Hashtable** contains an entry for each row of data. The entry contains the key, which is the row number, and the data value, which is yet another **Hashtable** whose key is the column number and whose data value is a **CommonValue** object containing the actual data. Remember that the **CommonValue** class provides us with the mechanism to store data and coerce it as requested by the application. If a column is null, we simply cannot store any information in the **Hashtable** for that column number.

After some sanity checking to ensure that we really need to look for the catalog information, we get a list of all of the tables. The **getTables** method in the **Connection** class provides us with a list of all of the SimpleText data files:

```
public Hashtable getTables(
    String dir,
    String table)
{
    Hashtable list = new Hashtable();

    // Create a FilenameFilter object. This object will only allow
    // files with the .SDF extension to be seen.
    FilenameFilter filter = new SimpleTextEndsWith(
                SimpleTextDefine.DATA_FILE_EXT);


    File file = new File(dir);

    if (file.isDirectory()) {
```

```
        // List all of the files in the directory with the .SDF extension
        String entries[] = file.list(filter);
        SimpleTextTable tableEntry;

        // Create a SimpleTextTable entry for each, and put in
        // the Hashtable.
        for (int i = 0; i < entries.length; i++) {

            // A complete driver needs to further filter the table
            // name here.
            tableEntry = new SimpleTextTable(dir, entries[i]);
                list.put(new Integer(i), tableEntry);
        }
    }

    return list;
}
```

Again, I use a **Hashtable** for each table (or file in our case) that is found. By now, you will have realized that I really like using **Hashtables**; they can grow in size dynamically and provide quick access to data. And because a **Hashtable** stores data as an abstract **Object**, I can store whatever is necessary. In this case, each **Hashtable** entry for a table contains a SimpleTextTable object:

```
public class SimpleTextTable
    extends        Object
{
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // Constructor
    //- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    public SimpleTextTable(
        String dir,
        String file)
    {
        this.dir = dir;
        this.file = file;

        // If the filename has the .SDF extension, get rid of it
        if (file.endsWith(SimpleTextDefine.DATA_FILE_EXT)) {
            name = file.substring(0, file.length() -
                        SimpleTextDefine.DATA_FILE_EXT.length());
        }
        else {
```

```
            name = file;
        }
    }

    public String dir;
    public String file;
    public String name;
}
```

Notice that the constructor strips the file extension from the given file name, creating the table name.

Now, back to the **getTables** method for **DatabaseMetaData**. Once a list of all of the tables has been retrieved, the **Hashtable** used for storing all of the rows is generated. If you were to add additional filtering, this is the place that it should be done. Finally, a new **ResultSet** object is created and initialized. One of the constructors for the **ResultSet** class accepts two **Hashtables**: one for the column information (**SimpleTextColumn** objects), and the other for row data (**CommonValue** objects). We'll see later how these are handled by the **ResultSet** class. For now, just note that it can handle both in-memory results (in the form of a **Hashtable**) and results read directly from the data file.

Statement

The **Statement** class contains methods to execute SQL statements directly against the database and to obtain the results. A **Statement** object is created using the **createStatement** method from the **Connection** object. Of note in Listing 10.21 are the three methods used to execute SQL statements: **executeUpdate**, **executeQuery**, and **execute**. In actuality, you only need to worry about implementing the **execute** method; the other methods use it to perform their work. In fact, the code provided in the SimpleText driver should be identical for all JDBC drivers.

Listing 10.21 Executing SQL statements.

```
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// executeQuery - JDBC API
// Execute an SQL statement that returns a single ResultSet.
//
//    sql    Typically this is a static SQL SELECT statement.
```

```
//
// Returns the table of data produced by the SQL statement.
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
public ResultSet executeQuery(
    String sql)
    throws SQLException
{
    if (traceOn()) {
        trace("@executeQuery(" + sql + ")");
    }

    java.sql.ResultSet rs = null;

    // Execute the query. If execute returns true, then a result set
    // exists.
    if (execute(sql)) {
        rs = getResultSet();
    }
    else {        // If the statement does not create a ResultSet, the
                  // specification indicates that an SQLException should
                  // be raised.
        throw new SQLException("Statement did not create a ResultSet");
    }
    return rs;
}

//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// executeUpdate - JDBC API
// Execute an SQL INSERT, UPDATE, or DELETE statement. In addition,
// SQL statements that return nothing, such as SQL DDL statements,
// can be executed.
//
//    sql    an SQL INSERT, UPDATE, or DELETE statement, or an SQL
//        statement that returns nothing.
//
// Returns either the row count for INSERT, UPDATE, or DELETE; or 0
// for SQL statements that return nothing.
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
public int executeUpdate(
    String sql)
    throws SQLException
{
    if (traceOn()) {
        trace("@executeUpdate(" + sql + ")");
    }
```

```
    int count = -1;

    // Execute the query. If execute returns false, then an update
    // count exists.
    if (execute(sql) == false) {
        count = getUpdateCount();
    }
    else {
        // If the statement does not create an update count, the
        // specification indicates that an SQLException should be raised.
        throw new SQLException("Statement did not create an update
          count");
    }

    return count;
}
```

As you can see, **executeQuery** and **executeUpdate** are simply helper methods for an application; they are built completely upon other methods contained within the class. The **execute** method accepts an SQL statement as its only parameter, and will be implemented differently, depending upon the underlying database system. For the SimpleText driver, the SQL statement will be parsed, prepared, and executed. Note that parameter markers are not allowed when executing an SQL statement directly. If the SQL statement created results containing columnar data, **execute** will return true; if the statement created a count of rows affected, **execute** will return false. If **execute** returns true, the application then uses **getResultSet** to return the current result information; otherwise, **getUpdateCount** will return the number of rows affected.

Warnings

As opposed to **SQLException**, which indicates a critical error, an **SQLWarning** can be issued to provide additional information to the application. Even though **SQLWarning** is derived from **SQLException**, warnings are not thrown. Instead, if a warning is issued, it is placed on a warning stack with the **Statement** object (the same holds true for the **Connection** and **ResultSet** objects). The application must then check for warnings after every operation using the **getWarnings** method. At first, this may seem a bit cumbersome, but when you consider the alternative of wrapping **try**...**catch** statements around each operation, this seems like a

better solution. Note also that warnings can be chained together, just like **SQLExceptions** (for more information on chaining, see the *JDBC Exception Types* section earlier in this chapter).

Two (Or More) For The Price Of One

Some database systems allow SQL statements that return multiple results (columnar data or an update count) to be executed. If you are unfortunate enough to be developing a JDBC driver using one of these database systems, take heart. The JDBC specification does address this issue. The **getMoreResults** method is intended to move through the results. Figuring out when you have reached the end of the results, however, is a bit convoluted. To do so, you first call **getMoreResults**. If it returns true, there is another **ResultSet** present and you can use **getResultSet** to retrieve it. If **getMoreResults** returns false, you have either reached the end of the results, or an update count exists; you must call **getUpdateCount** to determine which situation exists. If **getUpdateCount** returns -1, you have reached the end of the results; otherwise, it will return the number of rows affected by the statement.

The SimpleText driver does not support multiple result sets, so I don't have any example code to present to you. The only DBMS that I am aware of that supports this is Sybase. Because there are already multiple JDBC drivers available for Sybase (one of which I have developed), I doubt you will have to be concerned with **getMoreResults**. Consider yourself lucky.

PreparedStatement

The **PreparedStatement** is used for pre-compiling an SQL statement, typically in conjunction with parameters, and can be efficiently executed multiple times with just a change in a parameter value; the SQL statement does not have to be parsed and compiled each time. Because the **PreparedStatement** class extends the **Statement** class, you will have already implemented a majority of the methods. The **executeQuery**, **executeUpdate**, and **execute** methods are very similar to the **Statement** methods of the same name, but they do not take an SQL statement as a parameter. The SQL statement for the **PreparedStatement** was provided when the object was created with the **prepareStatement** method from the **Connection** object. One danger to note here: Because **PreparedStatement** is derived from the **State-**

ment class, all of the methods in **Statement** are also in **PreparedStatement**. The three execute methods from the **Statement** class that accept SQL statements are not valid for the **PreparedStatement** class. To prevent an application from invoking these methods, the driver should also implement them in **PreparedStatement**, as shown here:

```
// The overloaded executeQuery on the Statement object (which we
// extend) is not valid for PreparedStatement or CallableStatement
// objects.
public ResultSet executeQuery(
    String sql)
    throws SQLException
{
    throw new SQLException("Method is not valid");
}

// The overloaded executeUpdate on the Statement object (which we
// extend) is not valid for PreparedStatement or CallableStatement
// objects.
public int executeUpdate(
    String sql)
    throws SQLException
{
    throw new SQLException("Method is not valid");
}

// The overloaded execute on the Statement object (which we
// extend) is not valid for PreparedStatement or CallableStatement
// objects.
public boolean execute(
    String sql)
    throws SQLException
{
    throw new SQLException("Method is not valid");
}
```

Setting Parameter Values

The **PreparedStatement** class introduces a series of "set" methods to set the value of a specified parameter. Take the following SQL statement:

```
INSERT INTO FOO VALUES (?, ?, ?)
```

If this statement was used in creating a **PreparedStatement** object, you would need to set the value of each parameter before executing it. In the

SimpleText driver, parameter values are kept in a **Hashtable**. The **Hashtable** contains the parameter number as the key, and a **CommonValue** object as the data object. By using a **CommonValue** object, the application can set the parameter using any one of the supported data types, and we can coerce the data into the format that we need in order to bind the parameter. Here's the code for the **setString** method:

```
public void setString(
    int parameterIndex,
    String x)
    throws SQLException
{
    // Validate the parameter index
    verify(parameterIndex);

    // Put the parameter into the boundParams Hashtable
    boundParams.put(new Integer(parameterIndex), x);
}
```

The **verify** method validates that the given parameter index is valid for the current prepared statement, and also clears any previously bound value for that parameter index:

```
protected void verify(
    int parameterIndex)
    throws SQLException
{
    clearWarnings();

    // The paramCount was set when the statement was prepared
    if ((parameterIndex <= 0) ||
        (parameterIndex > paramCount)) {
        throw new SQLException("Invalid parameter number: " +
                                    parameterIndex);
    }

    // If the parameter has already been set, clear it
    if (boundParams.get(new Integer(parameterIndex)) != null) {
        boundParams.remove(new Integer(parameterIndex));
    }
}
```

Because the **CommonValue** class does not yet support all of the JDBC data types, not all of the set methods have been implemented in the SimpleText

driver. You can see, however, how easy it would be to fully implement these methods once **CommonValue** supported all of the necessary data coercion.

What Is It?

Another way to set parameter values is by using the **setObject** method. This method can easily be built upon the other set methods. Of interest here is the ability to set an **Object** without giving the JDBC driver the type of driver being set. The SimpleText driver implements a simple method to determine the type of object, given only the object itself:

```
protected int getObjectType(
    Object x)
    throws SQLException
{

    // Determine the data type of the Object by attempting to cast
    // the object. An exception will be thrown if an invalid casting
    // is attempted.
    try {
        if ((String) x != null) {
            return Types.VARCHAR;
        }
    }
    catch (Exception ex) {
    }

    try {
        if ((Integer) x != null) {
            return Types.INTEGER;
        }
    }
    catch (Exception ex) {
    }

    try {
        if ((byte[]) x != null) {
            return Types.VARBINARY;
        }
    }
    catch (Exception ex) {
    }

    throw new SQLException("Unknown object type");
}
```

Setting InputStreams

As we'll see with **ResultSet** later, using **InputStreams** is the recommended way to work with long data (blobs). There are two ways to treat **InputStreams** when using them as input parameters: Read the entire **InputStream** when the parameter is set and treat it as a large data object, or defer the read until the statement is executed and read it in chunks at a time. The latter approach is the preferred method because the contents of an **InputStream** may be too large to fit into memory. Here's what the SimpleText driver does with **InputStreams**:

```
public void setBinaryStream(
    int parameterIndex,
    java.io.InputStream x,
    int length)
    throws SQLException
{
    // Validate the parameter index
    verify(parameterIndex);

    // Read in the entire InputStream all at once. A more optimal
    // way of handling this would be to defer the read until execute
    // time, and only read in chunks at a time.
    byte b[] = new byte[length];

    try {
        x.read(b);
    }
    catch (Exception ex) {
        throw new SQLException("Unable to read InputStream: " +
                               ex.getMessage());
    }

    // Set the data as a byte array
    setBytes(parameterIndex, b);
}
```

But wait, this isn't the preferred way! You are correct, it isn't. The SimpleText driver simply reads in the entire **InputStream** and then sets the parameter as a byte array. I'll leave it up to you to modify the driver to defer the read until execute time.

ResultSet

The **ResultSet** class provides methods to access data generated by a table query. This includes a series of get methods which retrieve data in any one of the JDBC SQL type formats, either by column number or by column name. When the issue of providing get methods was first introduced by JavaSoft, some disgruntled programmers argued that they were not necessary; if an application wanted to get data in this manner, then the application could provide a routine to cross reference the column name to a column number. Unfortunately (in my opinion), JavaSoft chose to keep these methods in the API and provide the implementation of the cross reference method in an appendix. Because it is part of the API, all drivers must implement the methods. Implementing the methods is not all that difficult, but it is tedious and adds overhead to the driver. The driver simply takes the column name that is given, gets the corresponding column number for the column name, and invokes the same get method using the column number:

```
public String getString(
    String columnName)
    throws SQLException
{
    return getString(findColumn(columnName));
}
```

And here's the **findColumn** routine:

```
public int findColumn(
    String columnName)
    throws SQLException
{
    // Make a mapping cache if we don't already have one
    if (md == null) {
        md = getMetaData();
        s2c = new Hashtable();
    }

    // Look for the mapping in our cache
    Integer x = (Integer) s2c.get(columnName);

    if (x != null) {
        return (x.intValue());
    }
```

```
    // OK, we'll have to use metadata
    for (int i = 1; i < md.getColumnCount(); i++) {
        if (md.getColumnName(i).equalsIgnoreCase(columnName)) {

            // Success! Add an entry to the cache
            s2c.put(columnName, new Integer(i));
            return (i);
        }
    }

    throw new SQLException("Column name not found: " + columnName,
                           "S0022");
}
```

This method uses a **Hashtable** to cache the column number and column names.

It's Your Way, Right Away

An application can request column data in any one of the supported JDBC data types. As we have discussed before, the driver should coerce the data into the proper format. The SimpleText driver accomplishes this by using a **CommonValue** object for all data values. Therefore, the data can be served in any format, stored as a **CommonValue** object, and the application can request it in any other supported format. Let's take a look at the **getString** method:

```
public String getString(
    int columnIndex)
    throws SQLException
{
    // Verify the column and get the absolute column number for the
    // table.
    int colNo = verify(columnIndex);

    String s = null;

    if (inMemoryRows != null) {
        s = (getColumn(rowNum, columnIndex)).getString();
    }
    else {
        CommonValue value = getValue(colNo);

        if (value != null) {
            s = value.getString();
        }
    }
```

```
    if (s == null) {
        lastNull = true;
    }

    return s;
}
```

The method starts out by verifying that the given column number is valid. If it is not, an exception is thrown. Some other types of initialization are also performed. Remember that all **ResultSet** objects are provided with a **Hashtable** of **SimpleTextColumn** objects describing each column:

```
protected int verify(
    int column)
    throws SQLException
{
    clearWarnings();
    lastNull = false;

    SimpleTextColumn col = (SimpleTextColumn) inMemoryColumns.get(
                                    new Integer(column));

    if (col == null) {
        throw new SQLException("Invalid column number: " + column);
    }
    return col.colNo;
}
```

Next, if the row data is stored in an in-memory **Hashtable** (as with the **DatabaseMetaData** catalog methods), the data is retrieved from the **Hashtable**. Otherwise, the driver gets the data from the data file. In both instances, the data is retrieved as a **CommonValue** object, and the **getString** method is used to format the data into the requested data type. Null values are handled specially; the JDBC API has a **wasNull** method that will return true if the last column that was retrieved was null:

```
public boolean wasNull()
    throws SQLException
{
    return lastNull;
}
```

The SimpleText driver also supports **InputStreams**. In our case, the **SimpleTextInputStream** class is just a simple wrapper around a **CommonValue** object. Thus, if an application requests the data for a column as an **InputStream**, the SimpleText driver will get the data as a **CommonValue** object (as it always does) and create an **InputStream** that fetches the data from the **CommonValue**.

The **getMetaData** method returns a **ResultSetMetaData** object, which is our last class to cover.

ResultSetMetaData

The **ResultSetMetaData** class provides methods that describe each one of the columns in a result set. This includes the column count, column attributes, and the column name. **ResultSetMetaData** will typically be the smallest class in a JDBC driver, and is usually very straightforward to implement. For the SimpleText driver, all of the necessary information is retrieved from the **Hashtable** of column information that is required for all result sets. Thus, to retrieve the column name:

```
public String getColumnLabel(
     int column)
     throws SQLException
{
    // Use the column name
    return getColumnName(column);
}

protected SimpleTextColumn getColumn(
    int col)
    throws SQLException
{
    SimpleTextColumn column = (SimpleTextColumn)
                        inMemoryColumns.get(new Integer(col));

    if (column == null) {
        throw new SQLException("Invalid column number: " + col);
    }

    return column;
}
```

Summary

We have covered a lot of material in this chapter, including the JDBC **DriverManager** and the services that it provides, implementing Java interfaces, creating native JDBC drivers, tracing, data coercion, escape sequence processing, and each one of the major JDBC interfaces. This information, in conjunction with the SimpleText driver, should help you to create your own JDBC driver without too much difficulty.